BUNDESREPUBLIK DEUTSCHLAND

EP04/330A

19.04.2004

Prioritätsbescheinigung über die Einreichung einer Patentanmeldung

Aktenzeichen: 103 14 831.0

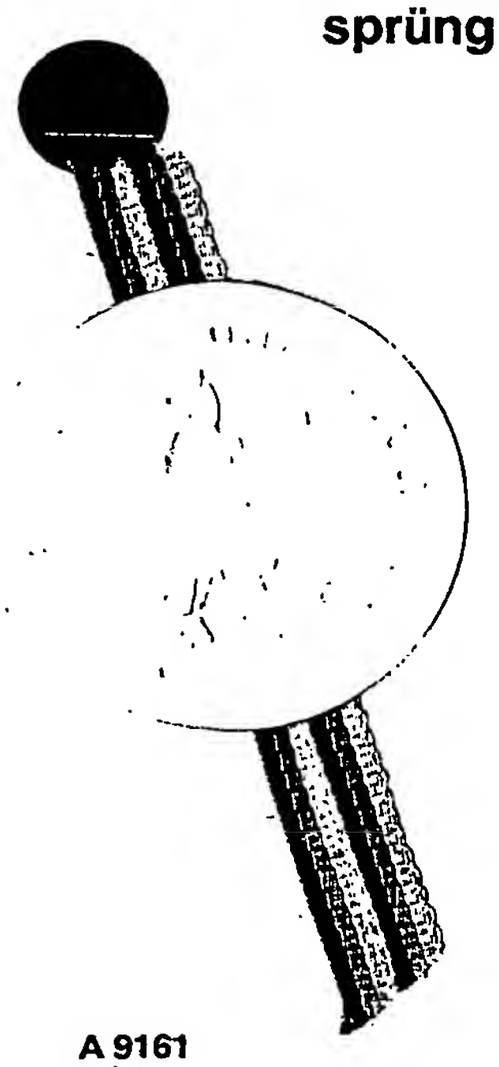
Anmeldetag: 01. April 2003

Anmelder/Inhaber: Siemens Aktiengesellschaft,
80333 München/DE

Bezeichnung: Verfahren und Anordnung zur Transformation von Quellcode

IPC: G 06 F 17/50

Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Patentanmeldung.



München, den 15. April 2004
Deutsches Patent- und Markenamt
Der Präsident
Im Auftrag

Sieck

A 9161
06/00
EDV-L

Beschreibung

Verfahren und Anordnung zur Transformation von Quellcode

Die Erfindung betrifft ein Verfahren und eine Anordnung zur Transformation von Quellcode, bei dem/der ein Quellcode, beispielsweise ein Java-Quellcode, in eine Darstellung in einer Meta-Auszeichnungssprache, beispielsweise XML, überführt, dort, beispielsweise mit XSLT, transformiert und dann diese in der Meta-Auszeichnungssprache formulierte transformierte Darstellung in einen modifizierten Quellcode, beispielsweise derselben Ausgangssprache, zurückverwandelt wird.

Aus dem Internet ist unter http://beautyj.berlios.de/ ein Java Source Code Transformation Tool BeautyJ bekannt, bei dem ein Java Quellcode in eine XML-Darstellung umgewandelt wird, mittels Sourclet API, beispielsweise durch Einfügen von Leerzeichen oder geänderten Kommentaren an bestimmten Stellen, „verschönert" und anschließend der modifizierte Quellcode in Java Quellcode zurück konvertiert werden kann. Eine Transformation mittels XSLT wird hier, für diesen Zweck, nur vorgeschlagen, aber nicht umgesetzt.

Die der Erfindung zugrunde liegende Aufgabe liegt nun darin, ein Verfahren und eine Anordnung zur Modifikation von Quellcode anzugeben, bei dem/der eine weitergehende noch flexiblere und effizientere Modifikation der Quellcodes erreicht wird.

HYPERLINK

Diese Aufgabe wird hinsichtlich des Verfahrens durch die Merkmale des Patentanspruchs 1 und hinsichtlich der Anordnung durch die Merkmale des Anspruchs 10 erfindungsgemäß gelöst.

Die weiteren Ansprüche betreffen bevorzugte Ausgestaltungen der Erfindung.

Die Erfindung besteht im Wesentlichen darin, dass der in eine Meta-Auszeichnungssprache, beispielsweise XML, transformierte Quellcode mit einer in seinen Elementen standardisierten und übersichtlich beschreibbaren Transformation, beispielsweise XSLT, derart transformiert wird, dass, nach einer Rückkonvertierung aus XML in die ursprüngliche Programmiersprache, ein neuer Quellcode entsteht, bei dem nicht nur die Darstellung, sondern auch der eigentliche Programminhalt bzw. die Funktionalität entsprechend den Transformationsvorschriften verändert wurde. Hierzu werden nur Transformationsregeln **TR** verwendet, die aus Bedingungen **C** und/oder Logik **L** und/oder Code-Fragementen **CF** bestehen, die mit Hilfe einer Transformation **T** zu einer Modifikation von **CodeML** in **CodeML*** führen, wodurch der Ausgangscode bspw. um eine Logging-Funktionalität oder eine Migrierbarkeits-Funktionalität ergänzt wird.

Die Erfindung wird im Folgenden anhand von in den Zeichnungen dargestellten Beispielen näher erläutert. Dabei zeigt

Figure 1 ein Gesamtblockdiagramm zur Erläuterung der Erfindung,

Figure 2 ein Blockschaltbild zur Erläuterung der erfindungsgemäßen Modifikation durch die Verwendung von Aspekten,

Figure 3 ein Blockschaltbild zur Erläuterung des erfindungsgemäßen Einfügens von Migrierbarkeits-Funktionalität und

Figure 4 ein Blockschaltbild zur Erläuterung der erfindungsgemäßen Modifikation durch den Einsatz von Templates, Filtern und Patterns.

In **Figure 1** ist ein Gesamtblockdiagramm zur Erläuterung der Erfindung dargestellt, bei dem zunächst ein Quellcode **SC**

durch einen Konverter **CONV** in einen in einer Meta-Auszeichnungssprache formulierten ersten Code **CodeML** umwandelt wird, wobei der Quellcode **SC** bei sofortiger Kompilierung einen Byte Code bzw. Binärcode **B** ergeben würde. Der in der Meta-Auszeichnungssprache dargestellte Code **CodeML** wird nun auf dem Wege einer Transformation **T** ausschließlich durch die Verwendung von Transformationsregeln **TR** modifiziert, welche aus Bedingungen **C** und/oder Logik **L** und/oder Code-Fragmenten **CF** bestehen, wodurch sich ein zweiter ebenfalls in der Meta-Auszeichnungssprache formulierten Code **CodeML*** ergibt. Ein weiterer Konverter **RCONV** wandelt nach der Transformation den Code **CodeML*** in einen Quellcode **SC*** zurück, der typischerweise in derselben Sprache wie der Quellcode **SC** formuliert ist. Durch einen Compiler **COMP** wird schließlich der modifizierte Code **SC*** in einen modifizierten Byte Code **B*** oder aber gleich in einen ausführbaren Binärcode umgewandelt. Wesentlich ist hierbei, dass sich der Byte-Code **B*** prinzipiell vom Byte-Code **B** unterscheidet bzw. dass der Quellcode nicht nur in seiner Darstellung, sondern auch in seinem Programmablauf geändert wurde.

Der Quellcode **SC** und der modifizierte Quellcode **SC*** sind beispielsweise in der Programmiersprache Java und die Codes **CodeML** und **CodeML*** sind beispielsweise in der Meta-Auszeichnungssprache XML formuliert, wobei „XML" für Extended Markup Language steht.

Die Transformation T, z. B. eine Extended Stylesheet Language Transformation oder **XSLT,** wird durch Transformationsregeln TR, z. B. innerhalb von **XSL** (Extended Stylesheet Language) Dateien beschrieben, wobei bspw. die in XSL formulierten Regeln **TR** u.a. beschreiben wie der in XML-codierte Quellcode **CodeML** mit dem Code-Fragment **CF** kombiniert wird, um einen neuen modifizierten Quellcode **CodeML*** mit integriertem **CF,** oder eine Abwandlung davon, zu bilden, welcher nun

beispielsweise zusätzliche Logging-Funktionalität enthalten kann.

In **Figure 2** ist ein erstes Ausführungsbeispiel dargestellt, bei dem die Transformationsregeln **TR** speziell Aspektregeln **AR** nach Aspektorientierter Programmierung (AOP) entsprechen, die in der AspectJ-Sprache ausgedrückt mindestens einen Pointcut **PC** und/oder mindestens einen Advice-Type **AT** und/oder mindestens ein Advice-Body **AB** enthalten und in ihrer Reihenfolge den Bestandteilen aus **Figure 1** zugeordnet werden können.

Auf diese Weise kann eine (werkzeugunabhängige) sogenannte AOP realisiert werden, die im Vergleich zu anderen Lösungsvarianten, beispielsweise AspectJ, keinen zusätzlichen Overhead im generierten Code **CodeML*** erzeugt und nicht den üblichen Einschränkungen (extra Kompiler, Syntax, usw.) existierender Aspektsprachen unterliegt.

Als Aspekt bezeichnet man in AOP eine Einheit, die überkreuzende Anliegen(crosscutting concerns) z.B. ein Logging, modularisiert und an einer Stelle kapselt. Der entsprechende Code, der bisher mehrere Module durchzog, wird hierbei mit Hilfe eines einzigen Aspekts zusammengeführt.

Die im **Anhang** befindlichen Programmauflistungen **Listing 1** bis **Listing 5** zeigen dies an einem konkreten Beispiel, bei dem zunächst die in Listing 1 enthaltene Datei TestCalculator.java in eine XML-Darstellung TestCalculator.xjava konvertiert wird. In **Listing 3** erfolgt die Beschreibung eines Aspektes in Form einer Datei LoggingAspect.xsl, die alle notwendigen Transformationsregeln enthält und dafür sorgt, dass jede Methode, die ein „cal" in ihrem Namen trägt, gefunden wird und zu Beginn der Ausführung dieser Methode ein Druckbefehl **System.out.println(„calculate begin")** und am Ende der Ausführung dieser Methode ein

Druckbefehl **System.out.println(„calculate end")** eingesetzt wird.

Sollen z.B. in allen 151 Klassen eines Projektes, alle Methoden die dem Muster **„cal"** entsprechen, also z.B. public String calcValues() o. ä., dazu veranlasst werden, eine System-Ausgabe beim Eintritt und Austritt zu tätigen, so werden zunächst mit

```
match="*[(name()='curly')and(ancestor::method[contains(name,'cal')])]"
```

- alle Methoden mit dem „cal"- Muster ausgewählt, mit

```
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> begin"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
```

- ein Statement **"System.out.println(%Name der Methode% + " begin")"**, z.B. **System.out.println("calculate end")**, eingefügt, mit

```
<xsl:copy-of select="*" />
```

- der ursprünglichen Code der Methode eingefügt und mit

```
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> end"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
```

- ein Statement **"System.out.println(%Name der Methode% + " end")"**, z.B. **System.out.println("calculate end")** eingefügt.

Anstatt also in allen 151 Klassen eine entsprechende Logging-Ausgabe zu veranlassen, kann dies hier innerhalb eines Logging-Aspektes an einer Stelle erfolgen. Änderungen müssen so auch nur an einer Stelle vorgenommen werden.

Figure 3 betrifft ein zweites Anwendungsbeispiel der Erfindung, bei dem ebenfalls aus einem Quellcode **CodeML** durch die Transformation **T** ein transformierter Code **CodeML*** erzeugt wird, welcher nun einen Mechanismus zur Sicherung (OLD) bzw. Ermittlung (NEW) mindestens eines Zustandes für die gewünschte (Versions)Migration enthält. Die Transformationsregeln **TR** sind in diesem Fall derart ausgebildet, dass sie als Migrationsregeln **MR** bezeichnet werden können und neben **C** und **L** zusätzlich mind. ein Fragment, sogenannte Checkpoints **CP**, zur Generierung (**CP Write**) bzw. zum Einlesen (**CP Read**) von Zuständen (**CP Data**) enthalten, die eine Migration von einer älteren Version **B*OLD** auf eine neuere Version **B*NEW** ermöglichen.
Die für eine Migration erforderliche Formatkonvertierungen der zu übergebenden Systemzustände können hiermit ebenfalls berücksichtigt werden. Zukünftige Migrationen brauchen hierdurch nicht schon im Vorfeld berücksichtigt zu werden, wodurch der Testaufwand und diesbezügliche potenzielle Programmfehler in frühen Programmversionen vermieden werden. Durch eine Automatisierung der Migration werden menschliche Fehler vermieden, da die Migration wesentlich systematischer erfolgt.

In **Figure 4** ist ein drittes Ausführungsbeispiel in mehreren Untervarianten dargestellt, bei dem ebenfalls ein in XML-codierter Quellcode **CodeML** durch eine Transformation **T** in einen modifizierten **CodeML*** umgewandelt wird. Die Transformation **T** wird hier jedoch durch Transformationsregeln **TR** bewirkt, die in jeder Variante aus mindestens **C** und **L** bestehen und wie bei der Umsetzung von Templates **TP** zusätzlich mindestens ein Template-Fragment **TPF** enthalten, bspw. für die Umwandlung in eine **EJB** (Enterprise Java Bean)

und bei der Umsetzung von Patterns **P** mindestens ein Pattern-Fragment **PF** besitzen, bspw. für die Anwendung von Proxy, Factory oder Singleton Patterns. Bei der Realisierung von Filtern **FI** genügen **C** und **L**, da hier nur Code entfernt wird und so bspw. überflüssige Output-Statements oder Kommentare beseitigt werden können.

Durch die entsprechende Anwendung von **proxy patterns** können local calls in remote calls oder in ähnlicher Weise local classes in EJB-classes (Enterprise Java Beans) umgewandelt werden.

Umgekehrt kann mit Hilfe einer Transformation **T** und entsprechenden Regeln **TR** aus dem XML-codierten Quellcode **JavaML** oder einem Fragment dieses Codes auch ein gültiges Template **TP** generiert werden, das als Vorlage für anderen Quellcode anwendbar ist.

Die oben genannten Ausprägungen des erfindungsgemäßen Verfahrens können einzeln und in beliebiger Reihenfolge nacheinander erfolgen.

Durch das erfindungsgemäße Verfahren ergeben sich noch eine Reihe von zusätzlichen Vorteilen, wie beispielsweise:

1. Es können schnelle und flexible Änderungen im Quellcode vorgenommen werden.

2. Es ist nur ein System für Problemstellungen wie Patternanwendung, Migration, AOP, Filtering, etc. erforderlich und nicht eine Reihe verschiedener teilweise proprietärer Werkzeuge.

3. Das Verfahren basiert auf Standards wie XML und XSLT und ist hinsichtlich der Konvertierbarkeit in andere Programmiersprachen geringeren Beschränkungen unterworfen als andere Verfahren zur Modifikation von Quellcode.

4. Selbst für spezielle und komplizierte Quellcode-Modifikationen sind keine proprietären Speziallösungen erforderlich, sondern es können hierfür existierende Standards wie **XSLT, XPath** und **Xquery** genutzt werden.

5. Diese Art der Modifikation erlaubt die Erstellung von Hierarchien u.a. durch die Möglichkeit der Hintereinanderausführung (Pipelines) mehrerer Transformationen.

7. Die Transformationen können als allgemeine Transformationen für eine Wiederverwendung in XSLT-Dateien gespeichert werden, so daß Bibliotheken z.B. für bestimmte Abläufe entstehen können.

8. Es kann eine XML-Repräsentation des Quellcodes in einer XML-Datenbasis gespeichert und bei Bedarf mit Hilfe einer XSLT in einfacher Weise an die jeweiligen Kundenbedürfnisse angepasst werden.

9. Durch die Verwendung der Standards **XmlSchema** oder **DTD** oder entsprechende XSLTs kann der Code vorab (ohne Kompilierung), auf bestimmte Korrektheitsaspekte hin, überprüft (validiert) werden.

10. Übliche XML-Visualisierungstools Tools können zur einfachen Bearbeitung bzw. Visualisierung und Bestimmung von Beziehungen im Code verwendet werden.

11. Dauerhafte XML-basierte Programmbibliotheken, die XPath-Anfragen unterstützen, können die Wiederverwendung von Code durch besseres Auffinden eines Codes bzw. von Code-Fragmenten oder Templates verbessert werden.

Anhang

Listing 1: TestCalculator.java

```
public class TestCalculator{
   private int z;

   public void calculate(int x, int y){
       z = x+y;
   }
}
```

Listing 2: TestCalculator.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestCalculator</name>
    <block>
      <var>
         <modifiers><private/></modifiers><type><int/></type><name>z</name>
       </var>
      <method>
        <modifiers><public/></modifiers>
        <type><void/></type>
        <name>calculate</name>
        <params>
          <param><type><int/></type><name>x</name></param>
          <param><type><int/></type><name>y</name></param>
        </params>
        <curly>
          <expr>
            <a>
              <name>z</name>
              <plus><name>x</name><name>y</name></plus>
            </a>
          </expr>
        </curly>
      </method>
    </block>
  </class>
</java>
```

Listing 3: LoggingAspect.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- ************************************
     *** copy template    **
     ************************************-->
<xsl:template match="*">
   <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>

<!-- ************************************
       ***  logging aspect              **
   ************************************-->
<xsl:template match="*[(name()='curly')and(ancestor::method[contains(name,'cal')])]">
  <xsl:copy>
```

```
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> begin"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
<xsl:copy-of select="*" />
<expr>
 <paren>
  <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
  <exprs>
   <expr>
    <xsl:text>"</xsl:text><xsl:value-of select="../name"/><xsl:text> end"</xsl:text>
   </expr>
  </exprs>
 </paren>
</expr>
</xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Listing 4: TestCalculator*.xjava

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
  <class>
    <modifiers><public/></modifiers>
    <name>TestCalculator</name>
    <block>
      <var>
        <modifiers><private/></modifiers><type><int/></type><name>z</name>
      </var>
      <method>
        <modifiers><public/></modifiers>
        <type><void/></type>
        <name>calculate</name>
        <params>
          <param><type><int/></type><name>x</name></param>
          <param><type><int/></type><name>y</name></param>
        </params>
        <curly>
          <expr>
            <paren>
              <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
              <exprs><expr>"calculate begin"</expr></exprs>
            </paren>
          </expr>
          <expr>
            <a>
              <name>z</name>
              <plus><name>x</name><name>y</name></plus>
            </a>
          </expr>
          <expr>
            <paren>
              <dot><dot><name>System</name><name>out</name></dot><name>println</name></dot>
              <exprs><expr>"calculate end"</expr></exprs>
            </paren>
          </expr>
        </curly>
      </method>
    </block>
  </class>
</java>
```

Listing 5: TestCalculator*.java

```
public class TestCalculator{
   private int z;

   public void calculate(int x, int y){
        System.out.println("calculate begin");
        z = x+y;
        System.out.println("calculate end");
   }
}
```

Patentansprüche

1. Verfahren zur Transformation von Quellcode,
- bei dem ein in einer ersten Programmiersprache formulierter Quellcode (SC) in einen in einer Meta-Auszeichnungssprache formulierten ersten Code (CodeML) umgewandelt wird,
- bei dem eine Transformation (T) nur in Abhängigkeit von Transformationsregeln TR in einen in der Meta-Auszeichnungssprache formulierten zweiten Code (CodeML*) erfolgt und
- bei dem dieser zweite Code in einen in der ersten Programmiersprache oder einer anderen Programmiersprache formulierten zweiten Quellcode (SC*) verwandelt wird, wobei sich der erste und der zweite Quellcode in ihrer Funktionalität (B,B*) unterscheiden.

2. Verfahren nach Anspruch 1,
bei dem die Transformationsregeln (TR) mindestens eine Bedingung C und einen Logikbestandteil L und/oder Codefragment CF selbst enthalten.

3. Verfahren nach Anspruch 1 oder 2,
bei dem die Transformationsregeln TR als Aspektregeln AR nach AOP bezeichnet werden und/oder C und/oder L und/oder CF mindestens einen Pointcut PC und/oder mindestens einen Advice-Type AC und/oder mindestens einen Advice-Body AB bilden.

4. Verfahren nach Anspruch 1 oder 2,
bei dem die Transformationsregeln TR als Migrationsregeln MR bezeichnet werden und/oder C und/oder L und/oder CF mindestens einen Read und/oder Write Checkpoint in CodeML* unter Ausführung der Transformation T generieren.

5. Verfahren nach Anspruch 1 oder 2,
bei dem die Transformationsregeln TR mindestens ein Fragment in Form eines Templates TPF und/oder und/oder mindestens eines Patterns PF entalten, bei denen mit Hilfe der Transformation T mindestens eine oder keine Codeentfernung und/oder Codeveränderung und/oder Codeerzeugung im CodeML erfolgt, und sich aber mindestens ein Codeteil von CodeML* gegenüber CodeML ändert.

6. Verfahren nach Anspruch 1 oder 2,
bei dem in den Transformationsregeln TR kein Codefragment CF existiert und/oder C und/oder L als mindestens ein Filter benutzt werden, welche den nach der Transformtion T entstehenden CodeML* um mindestens einen elementaren Bestandteil bereinigen.

7. Verfahren nach den Ansprüchen 1 bis 5,
bei dem CF mindestens einen Programmteil zur schritthaltenden Dokumentation des Programmablaufs oder des Ablaufs von Programmteilen enthält, und dieser durch die Transformation T in gleicher oder abgewandelter Form an mindestens einer Stelle in CodeML* vorkommt.

8. Verfahren nach Anspruch 4,
bei dem TR derart ausgebildet ist, dass mit Hilfe der Transformationen T ein Mechanismus zur Sicherung mindestens eines Systemzustandes in den zweiten Quellcode (CodeML*) eingebracht wird, um eine Migration in andere Versionen zu ermöglichen.

9. Verfahren nach einem der vorhergehenden Ansprüche,
bei dem die Programmiersprache des ersten und zweiten Quellcodes Java und die Meta-Auszeichnungssprache XML ist und bei dem die Transformation mit XSLT erfolgt.

10. Verfahren nach Anspruch 1 oder 2,
bei dem mit Hilfe der Transformation T aus dem ersten Code oder einem Fragment des ersten Codes mindestens ein Template TP gebildet wird, welches im Rahmen des Anspruchs 5 verwendet werden kann.

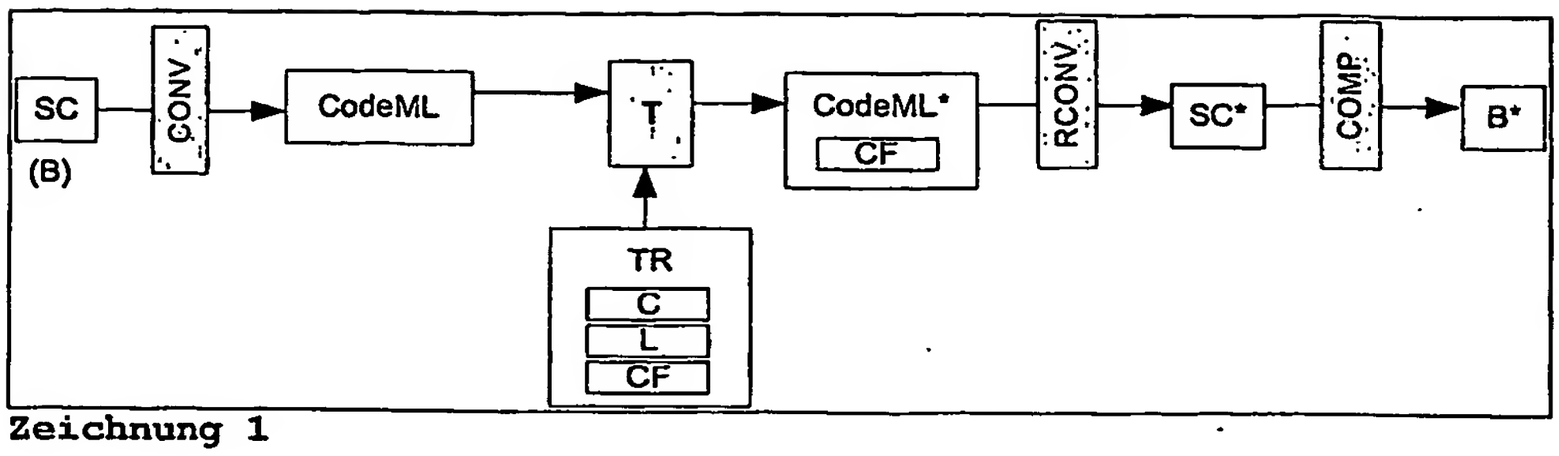
11. Anordnung zur Transformation von Quellcode,
- bei der ein erster Konverter (CONV) derart vorhanden ist, dass ein in einer ersten Programmiersprache formulierter Quellcode (SC) in einen in einer Meta-Auszeichnungssprache formulierten ersten Code (CodeML) umgewandelt wird,
- bei der ein Prozessor derart vorhanden ist, dass der CodeML durch eine Transformation (T) nur in Abhängigkeit von Transformationsregeln (TR) in einen in der Meta-Auszeichnungssprache formulierten zweiten Code (CodeML*) umgewandelt wird und
- bei der ein zweiter Konverter (RCONV) derart vorhanden ist, dass dieser zweite Code in einen in der ersten Programmiersprache oder einer anderen Programmiersprache formulierten zweiten Quellcode (CodeML*) verwandelt wird,
wobei sich der erste und der zweite Quellcode in ihrer Funktionalität bzw. Inhalt(B,B*) unterscheiden.

Zusammenfassung

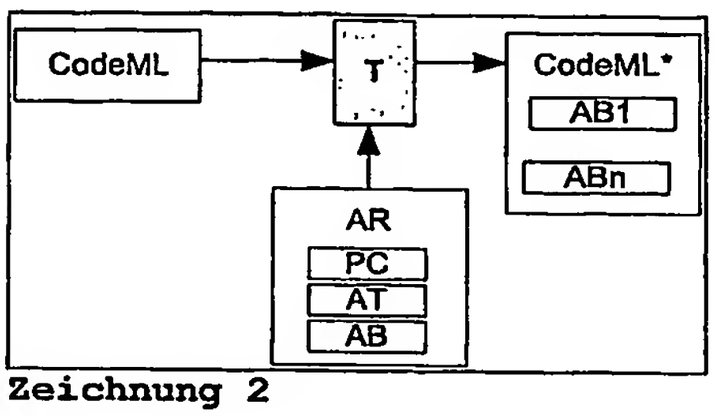
Verfahren und Anordnung zur Transformation von Quellcode

Die Erfindung besteht im Wesentlichen darin, dass der in eine Meta-Auszeichnungssprache, beispielsweise XML, konvertierte Quellcode mit einer in seinen Elementen standardisierten und übersichtlich beschreibbaren Transformation, beispielsweise XSLT, derart transformiert wird, dass, nach einer Rückkonvertierung aus XML in die ursprüngliche Programmiersprache, ein neuer Quellcode entsteht, bei dem nicht nur die Darstellung, sondern auch der eigentliche Programminhalt bzw. die Funktionalität entsprechend den Transformationsvorschriften verändert wurde. Hierbei sorgt nur die in den Transformationsvorschriften enthaltenen Regeln für eine Modifikation des ursprünglichen Quellcodes, wodurch beispielsweise der Code um Logging-Funktionalität oder eine Migrierbarkeits-Funktionalität ergänzt wird.

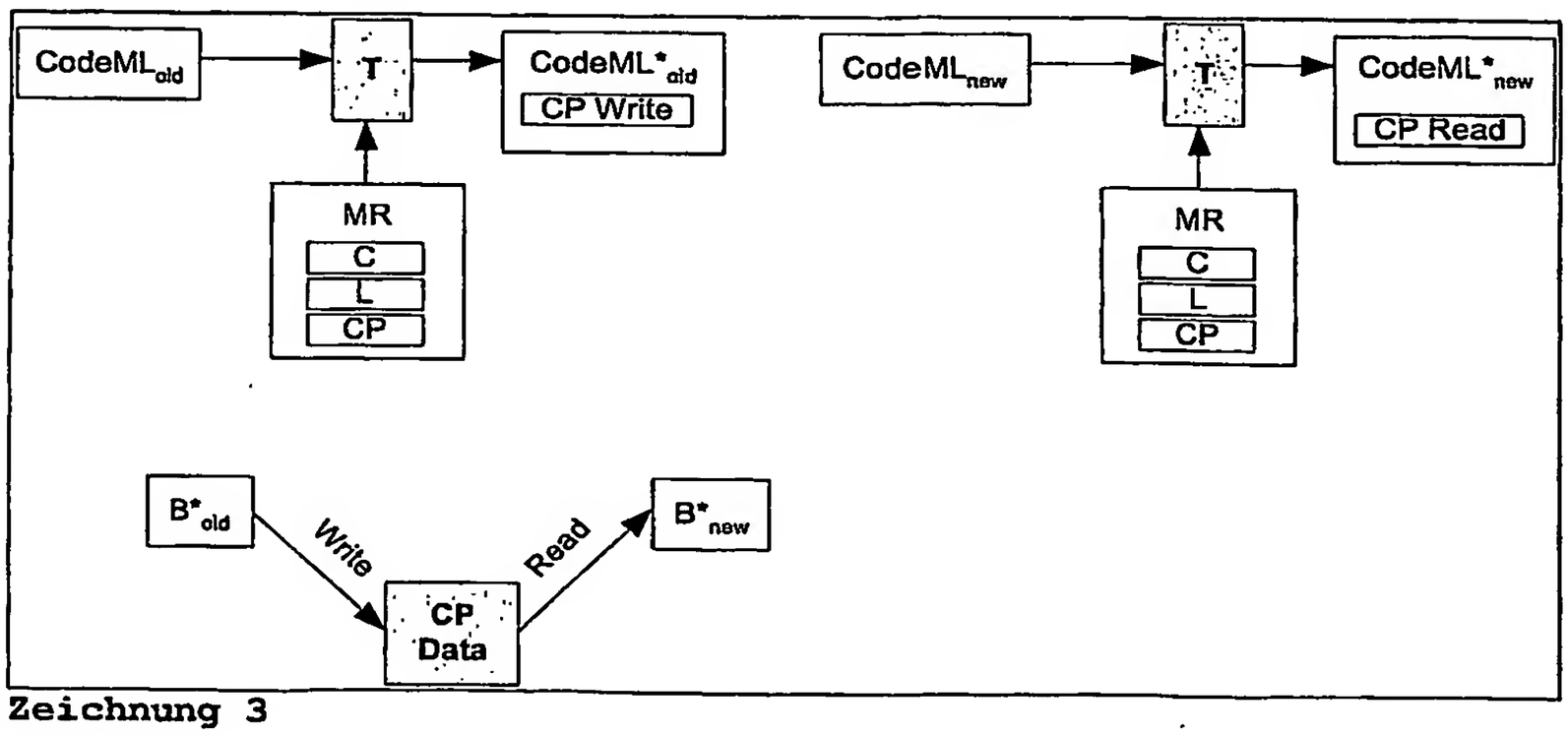
Figure 1



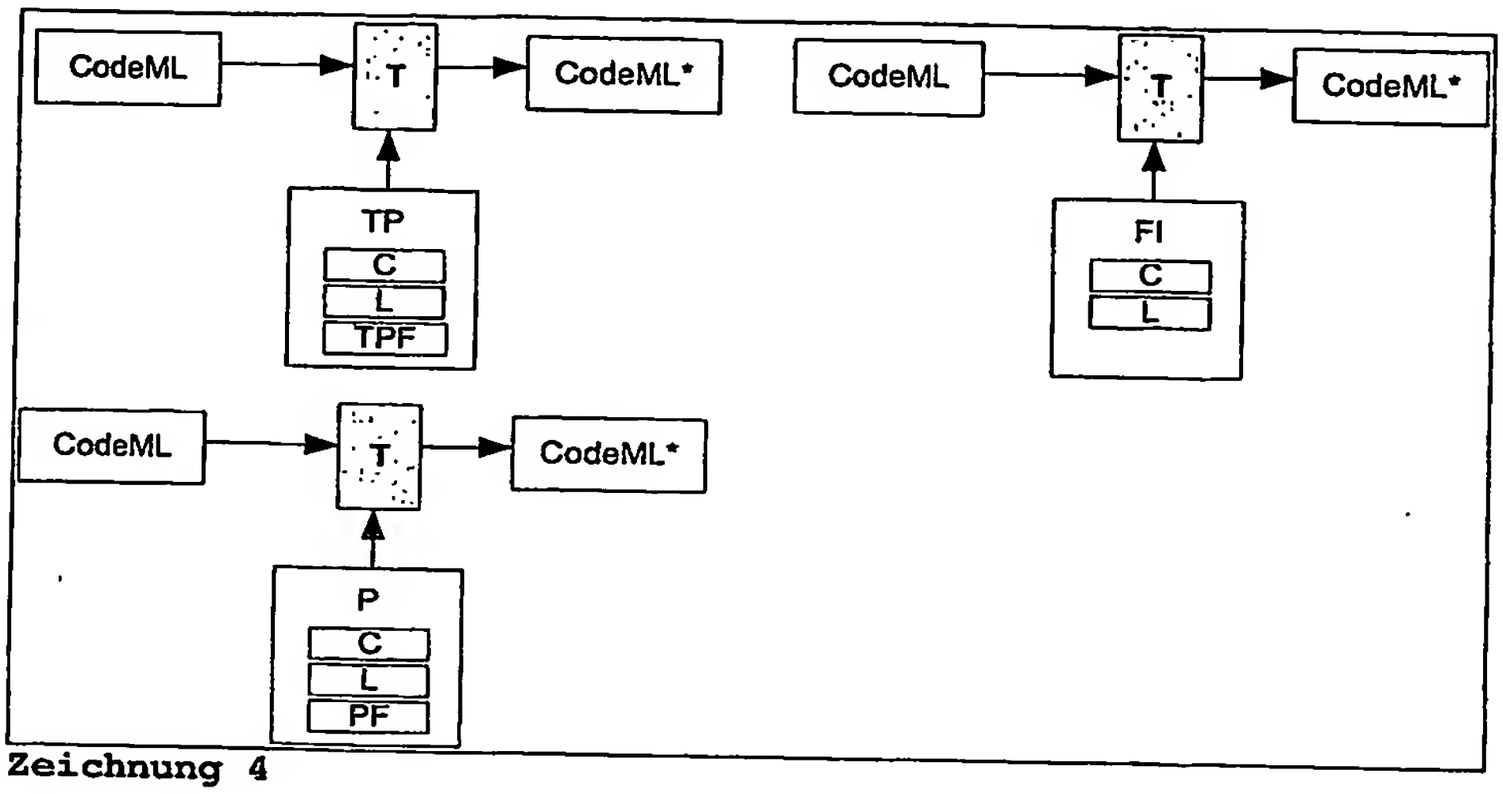
Zeichnung 1



Zeichnung 2



Zeichnung 3


CodeML
T
CodeML*
TP
C
L
TPF
CodeML
T
CodeML*
FI
C
L
CodeML
T
CodeML*
P
C
L
PF

Zeichnung 4